CHEP 2018
23RD INTERNATIONAL CONFERENCE ON
COMPUTING IN HIGH ENERGY AND NUCLEAR PHYSICS
9-13 July 2018
National Palace of Culture
Sofia, Bulgaria

STRATEGY FOR MULTITHREADING / VECTORIZATION

GERHARD RAVEN / VU AMSTERDAM & NIKHEF

CHEP 2018
23RD INTERNATIONAL CONFERENCE ON
COMPUTING IN HIGH ENERGY AND NUCLEAR PHYSICS
9-13 July 2018
National Palace of Culture
Sofia, Bulgaria

STRATEGY FOR PARALLELISM

GERHARD RAVEN / VU AMSTERDAM & NIKHEF

Declarative Control Flow

Prepared for CppCon

Bellevue, WA, Sep 20–25, 2015

Andrei Alexandrescu, Ph.D.
andrei@erdani.com

cppcon the c++ conference

Declarative Control Flow

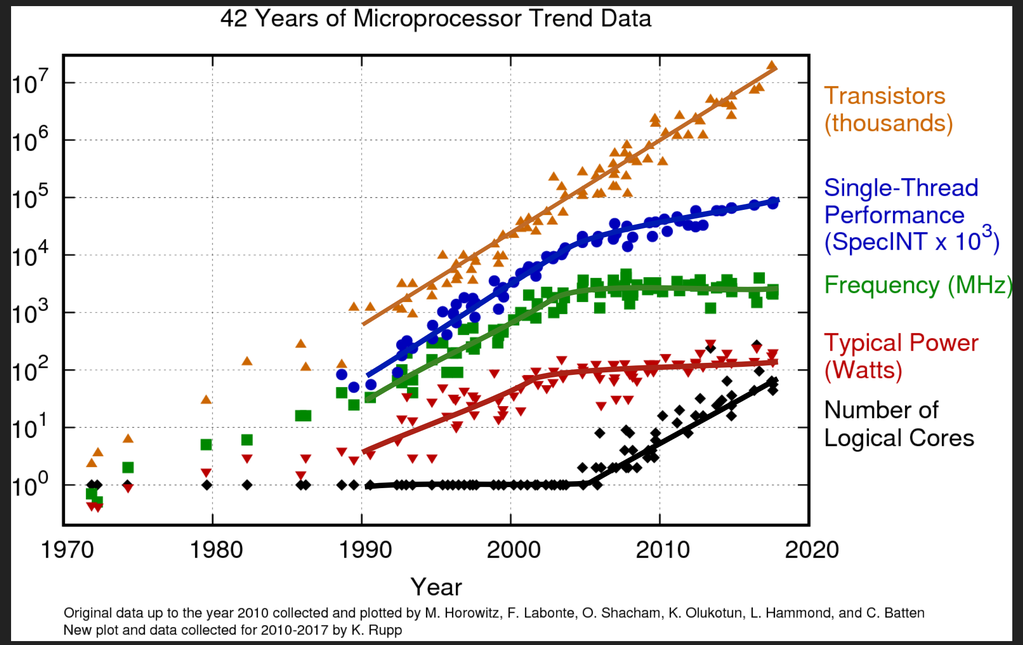
www.CppCon.org

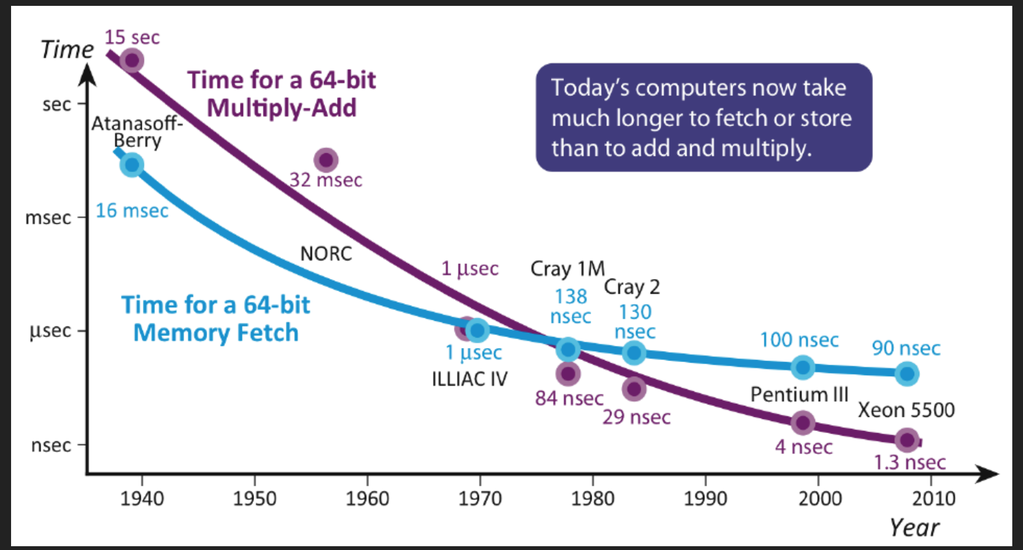
WHAT IS THE ~~PROBLEM~~ CHALLENGE

- Single-thread performance increase has stalled a decade ago…
- CPU / memory gap growing

In the mean time:

- Core count grows
- Vectorization is back! (did it ever go away?)
- Accelerators is where the growth is…

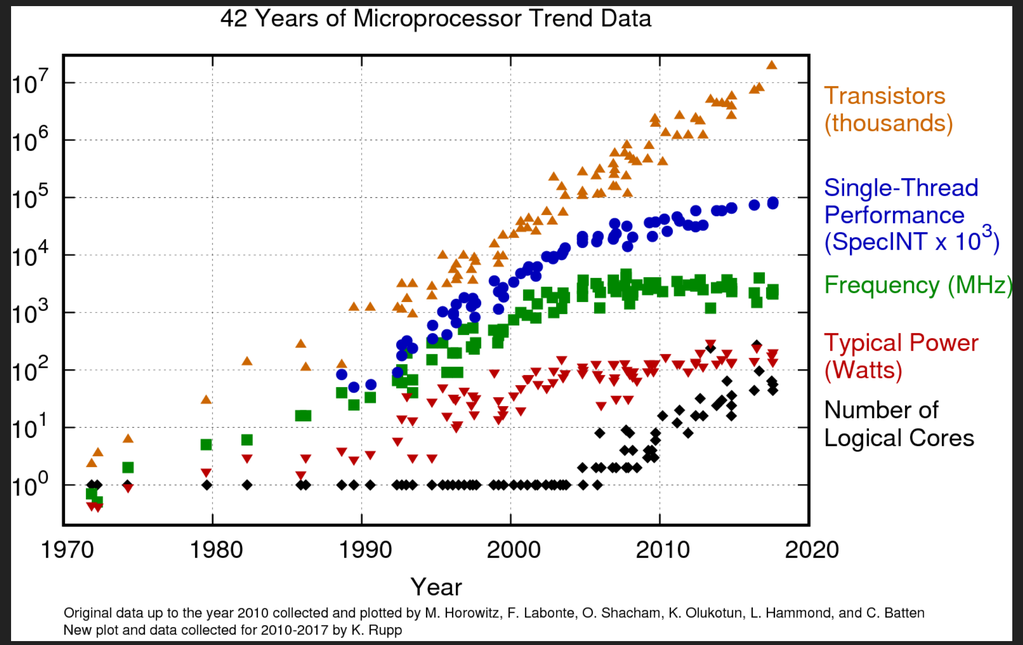


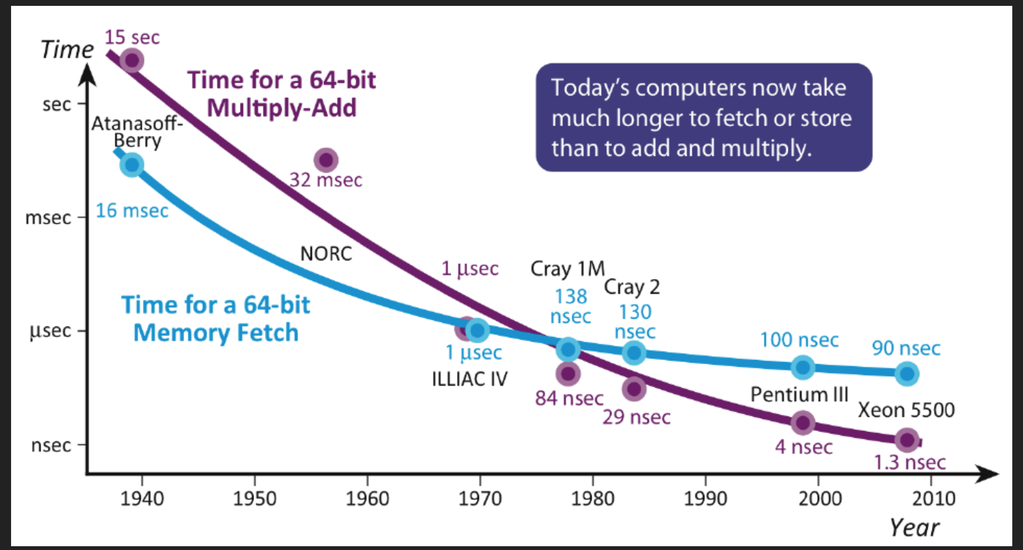


THE ~~PROBLEM~~ CHALLENGE

Taking advantage of new architectures and programming paradigms will be critical for HEP to increase the ability of our code to deliver physics results efficiently, and to meet the processing challenges of the future.

(A Roadmap for HEP Software and Computing R&D for the 2020s. arXiv:1712.06982

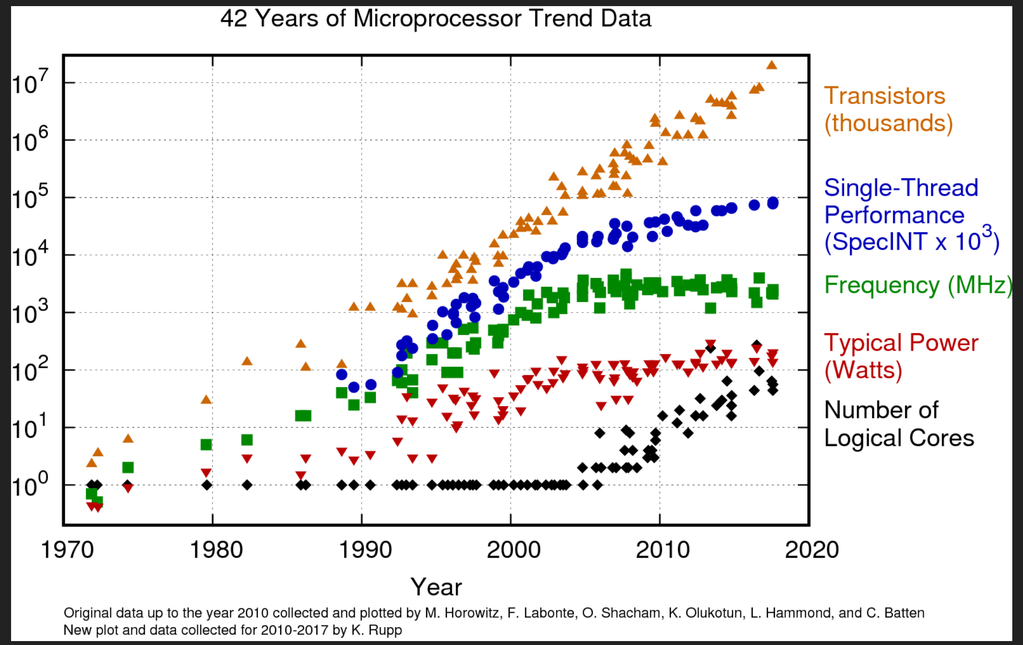


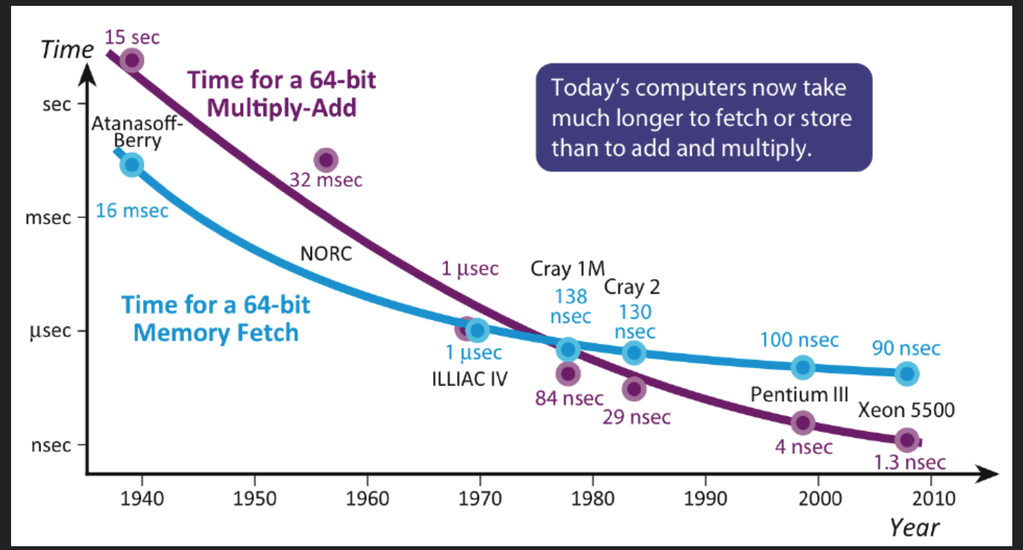


THE ~~PROBLEM~~ CHALLENGE

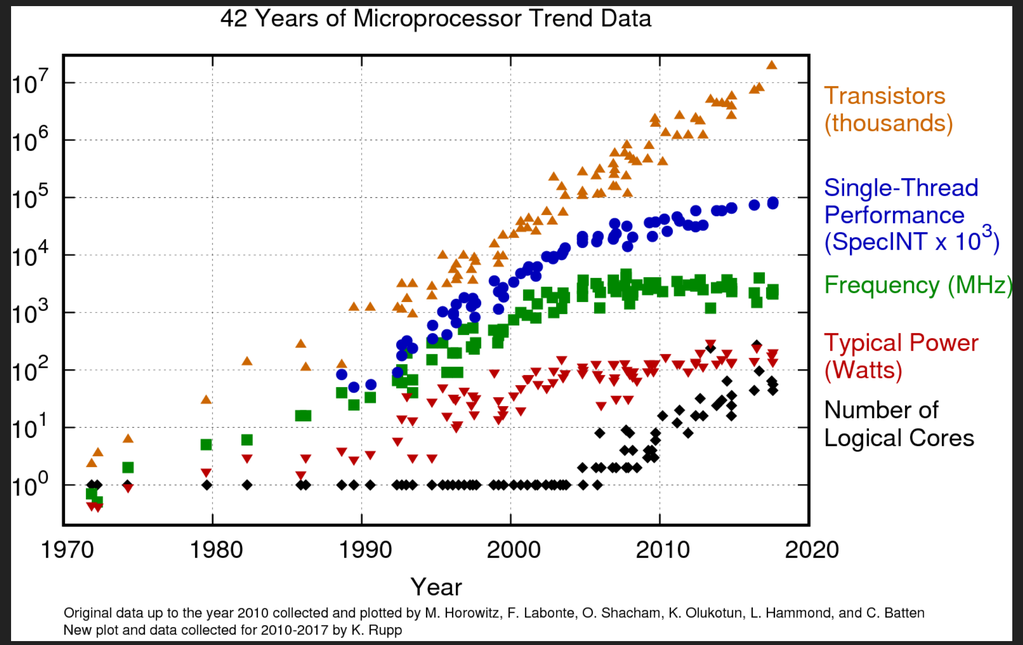
Taking advantage of new architectures and programming paradigms will be critical for HEP ~~to increase the ability of our code~~ to deliver physics results ~~efficiently, and to meet the processing challenges of the future.~~

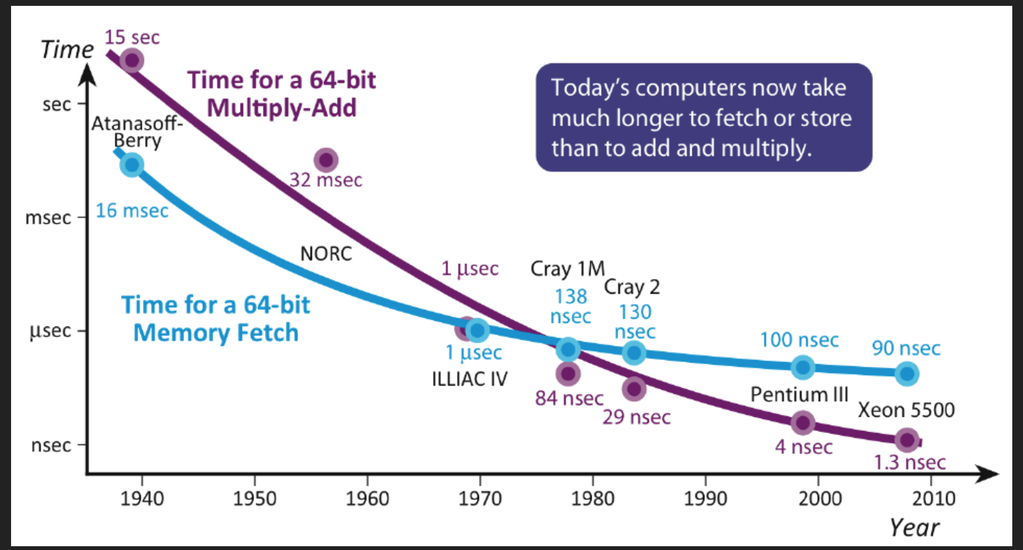
([A Roadmap for HEP Software and Computing R&D for the 2020s. arXiv:1712.06982](#))



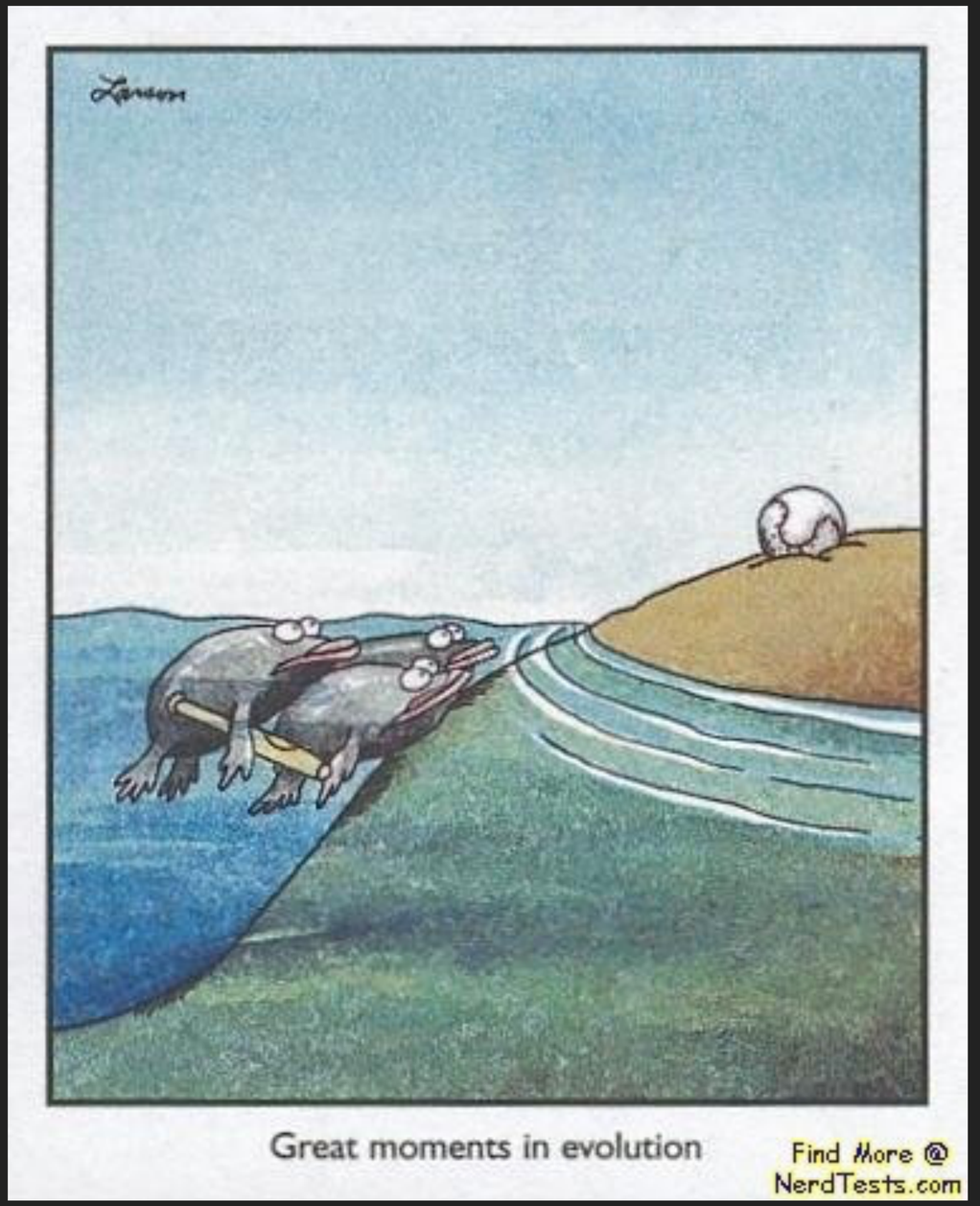


THE ~~PROBLEM~~ CHALLENGE



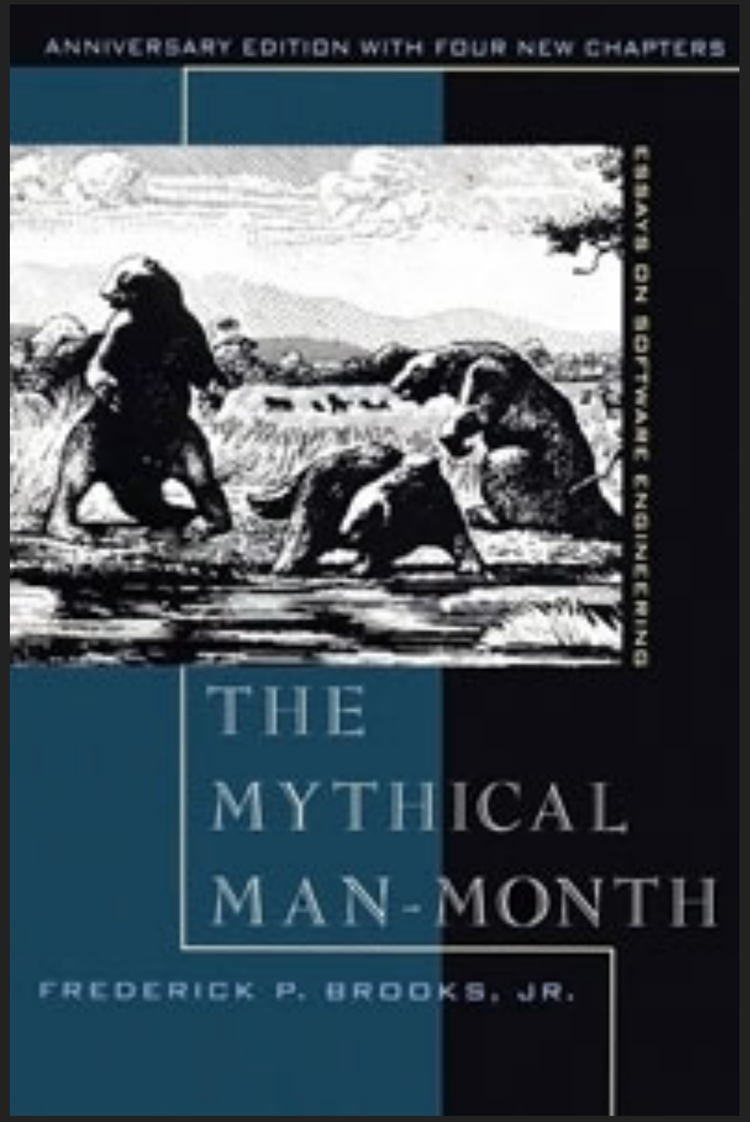


THE ~~PROBLEM~~ CHALLENGE



DESIGN PRINCIPLES: CONCEPTUAL INTEGRITY

- The design is coherent, reliable, and does what the user expects it to do.
  - **Be consistent:** Don't make similar things different, including in spelling, behavior, or capability. Don't make different things appear similar when they have different behavior or capability.
  - **Be orthogonal:** Avoid arbitrary coupling. Let features be used freely in combination.
  - **Be general:** Don't restrict what is inherent. Don't arbitrarily restrict a complete set of uses. Avoid special cases and partial features.



Brooks 1975

DESIGN GOAL

- “Be Minimal” – hard to abuse, easy to use right

DESIGN GOAL

- “Be Minimal” – hard to abuse, easy to use right

DESIGN GOAL

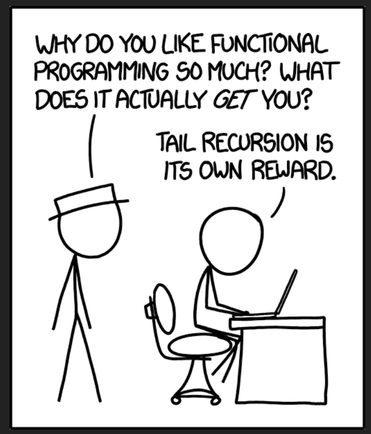
- “Be Minimal” – hard to abuse, easy to use right

“I choose a block of marble and chop off whatever I don't need.”

– Auguste Rodin

HOW: TAKING AWAY PRIMITIVES —— INTRODUCE ABSTRACTIONS

- Structured programming – Dijkstra: GOTO considered harmful
  - use loop constructs (for, while) instead
- Procedural programming – modularization
  - (local) scope
- Object-Oriented programming – dependency inversion
  - takes away ‘void *’, use VTBL instead – allows to call lower-level code ‘not yet written’
  - hide state
- Functional programming – takes away (mutable) state
  - powerful type systems, referential transparency
- Declarative programming – takes away control flow



DECLARATIVE INTERFACES & COMPOSITION

- Separate “what” from “how” & “when”
- Allows the underlying implementations to evolve / adapt
  - eg. ‘transpose’ traversal, enabling vectorization
- Gets rid of ‘boilerplate’
- Enables composition
- see eg. talks by Jim Pivarski, Enrico Gireaud, Gordon Watts

Improving on current interfaces

```
TTreeReader data(tree);
TTreeReaderValue<A> x(data, "x");
TTreeReaderValue<B> y(data, "y");
TTreeReaderValue<C> z(data, "z");

while (reader.Next()) {
   if (IsGoodEvent(x, y, z))
      DoStuff(x, y, z);
}
```

```
ROOT::EnableImplicitMT();
TDataFrame data(tree, {"x","y","z"});


data.Filter(IsGoodEvent)
    .Foreach(DoStuff);
```

- ● users have full control over the event-loop
- ✔ needs some boilerplate
- ✔ running the event-loop in parallel is not trivial
- ✔ users implement trivial operations again and again

5

E. Gireaud & D. Piparo, ROOT interface discussion 1/3/2017

COMPOSING COMPONENTS → PIPELINES & GRAPHS

C++ now 2018 MAY 7-11 cppnow.org

Ben Deane

Easy to Use,
Hard to Misuse
Declarative Style in C++

"WHOLEMEAL PROGRAMMING"

Declarative style is about processing data pipelines.

When you have composable pieces, rearranging and exploring data is quick and easy.

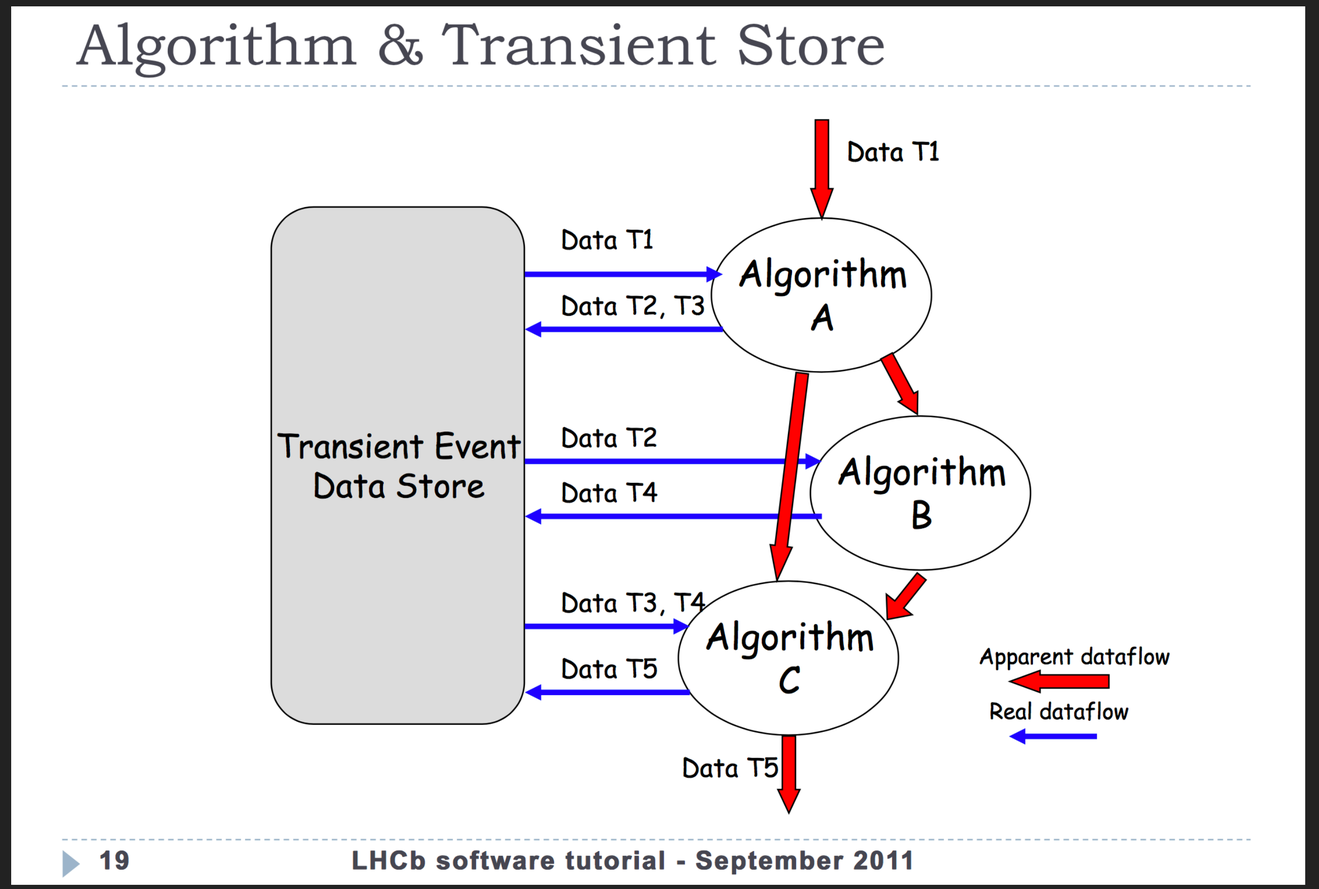
Compare: unix command-line.

- generators (find, `iota`)
- selections (grep, `unique`)
- transformations (cut, tr, `transform`)
- permutations (sort, `shuffle`)
- reductions/unfolds (wc, xargs, `accumulate`)

82 / 122

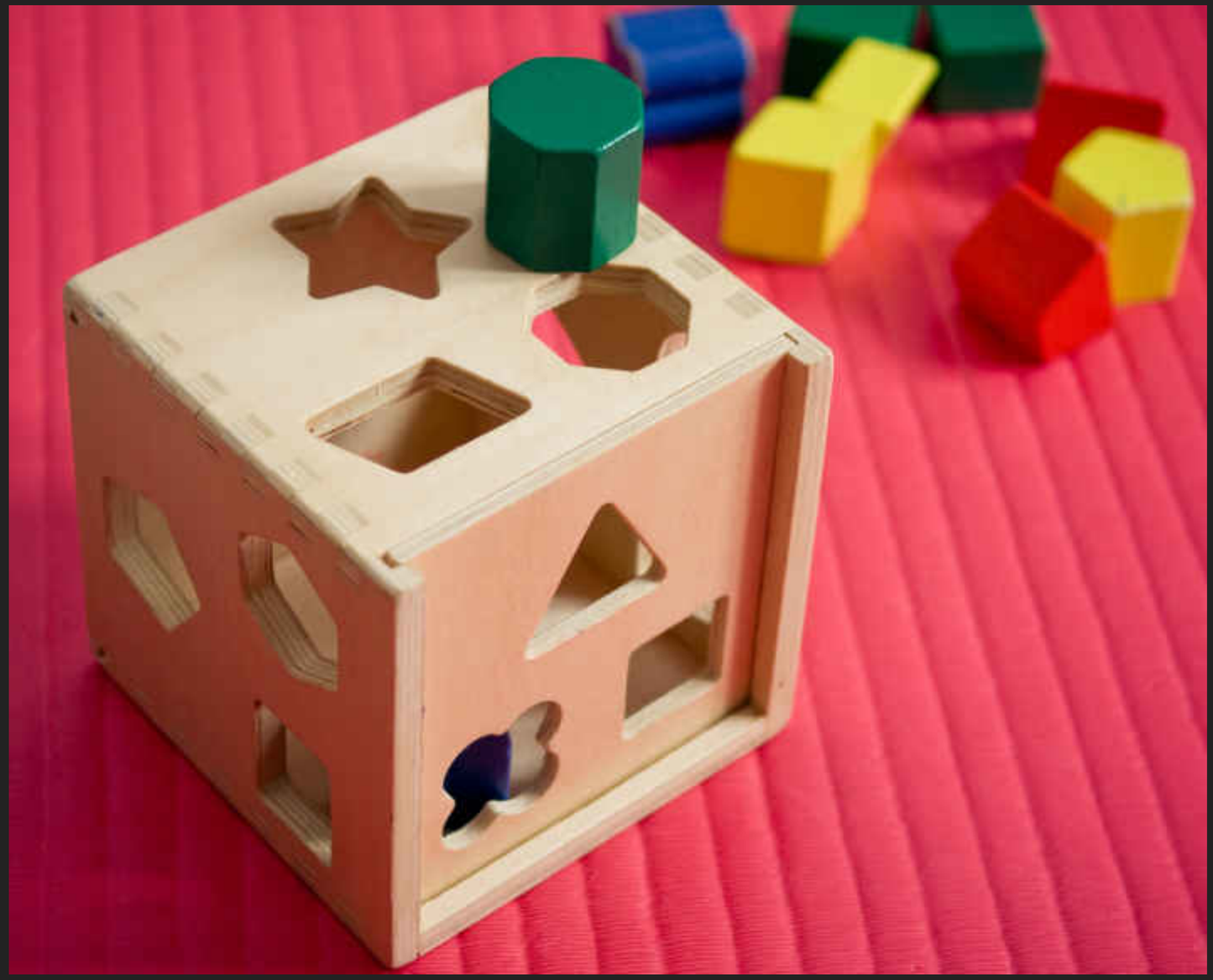
COMPONENT INTERFACES AND FRAMEWORK ABSTRACTIONS…

- User code *should not care* how to obtain its data
- If the user code does not have specify *how*, the ‘back-end’ can be changed without changing the ‘business code’
  - Avoids ‘magic incantations’ which need to be documented / taught / enforced / updated
  - what if multiple events ‘in-flight’? – must talk to the *relevant* data store
  - what if the data store is replaced by a messaging system?
    - note: must pay attention to the assumed lifetime of the *input* – but that is what ‘const&’ and ‘&&’ are for…
- Distinguish between [plumbing and porcelain]



DECLARATIVE COMPONENTS

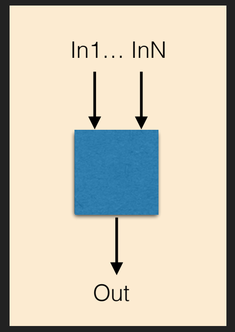
- Make the input/output part of the component interface signature
- Example of a ‘minmal, generic’ interface



DECLARATIVE COMPONENTS

- Make the input/output part of the component interface signature
- Example of a ‘minmal, generic’ interface
- Today, uses an “event data store” – but the user code doesn’t know/care!
  - No “user changes” if the ‘data store’ is replaced with eg. ‘message queue’ (0MQ, MPI, …)
    - modulo small print on (expectations about) lifetime of the input (in case the output wants to reference back to the input!) – but that can be clarified by tweaking the input types

Example: *Gaudi::Functional::Transformer*




```
template <typename Out, typename... In>
class Transformer<Out(In const&…)>
{
   virtual Out operator()(In const&...) const = 0;
};
```

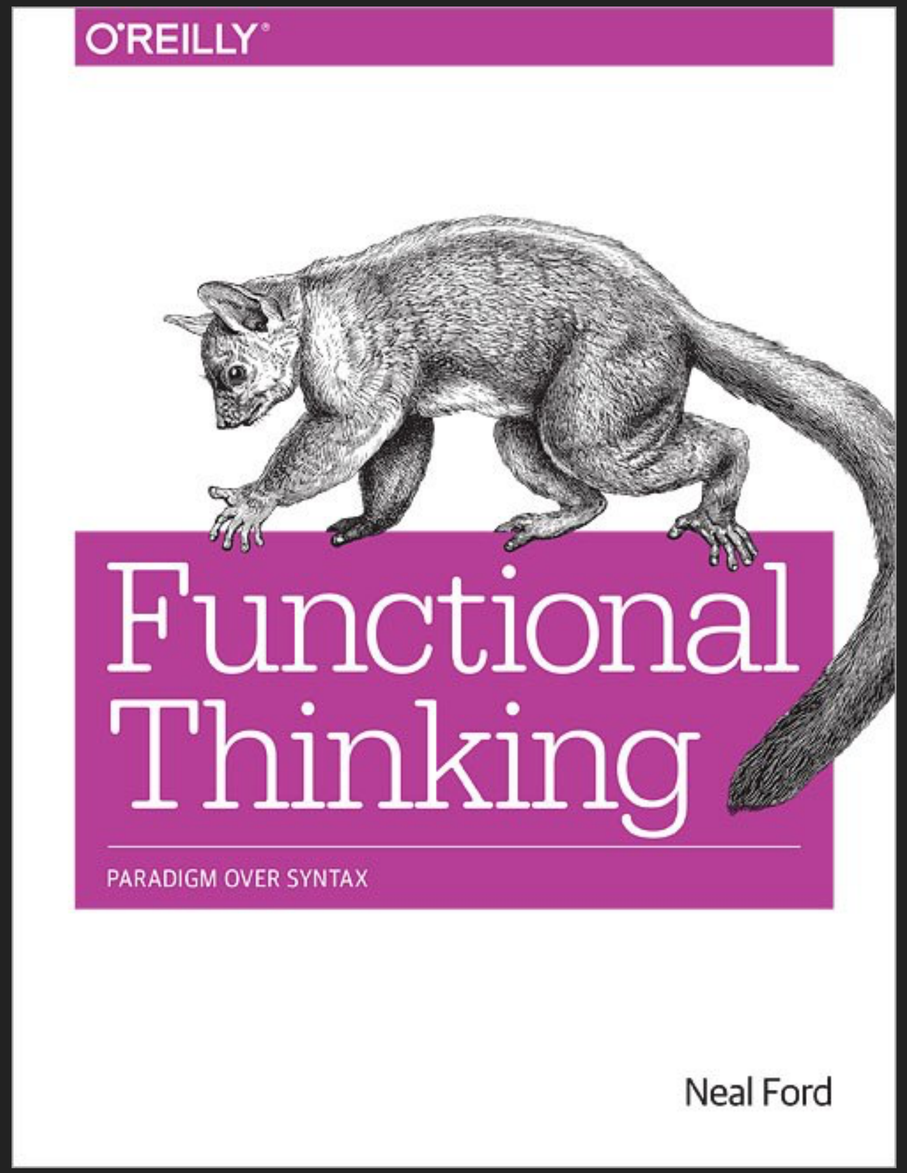
declarative: states what it requires as input, produces as output

```
class ElectronFinder : Transformer<Electrons(Tracks const&, Clusters const&)>
{
   Electrons operator()(Tracks const&, Clusters const&) const override
};
```

const: makes it hard(er) to write code which misbehaves in a multithreaded environment!
Check Herb Sutter's presentation for much more information

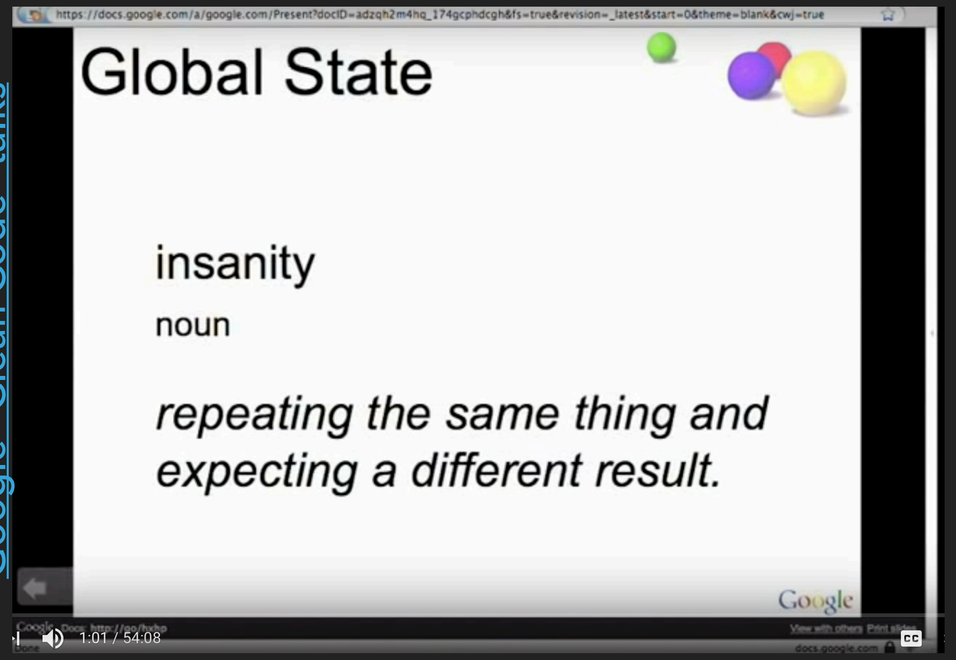
IMMUTABLE COMPONENTS

- Once initialized, components should be immutable
  - Small print: state of counters/histograms does not affect processing –- so does not matter
  - Easier to understand & test
  - Re-entrant – which is necessary for scaling on massive parallel systems
- All ‘varying state’ *explicitly* passed as arguments
  - A must for offloading to an accelerator



WHAT ABOUT THE EVENT DATA?

Google "Clean Code" talks



	Shared	Not Shared
Mutable	XXX	~
Immutable	VVV	VVV

- Parallel processing + shared mutable data → race conditions + non-determinism

IMMUTABLE DATA — ONCE SHARED

- Avoid large ‘objects’ which need to be modified
- Instead, compose smaller pieces, which individually are immutable
  - Think ‘zip’ (python) or ‘join’ (SQL) – using thin proxies, which are transparant to the compiler, and give the *illusion* of objects containing just what is needed, nothing more
- Admittedly, some things become more difficult (eg. bi-directional links)

LIFTING: ABSTRACTING "OUTER LOOP PARALLELISM"

- "Lifting is a concept which allows you to transform a function into a corresponding function within another (more general) setting"
- User provides `*Scalar1→ Scaler2*`, Framework uses CRTP to "lift" it to a `*Vector<Scalar1> → Vector<Scalar2>*` transformer
  - this is python's map
- Could use *std::transform*, TBB, libDispatch, *std::async* (don't!), Parallel STL, … to dispatch the work – and decide eg. on 'chunk size'
  - note: as the output needs to be 'gathered', anything but *std::transform* only useful if 'lot of work' compared to the gathering overhead

CRTP

```
struct FromPrTrack_
: ScalarTransformer<FromPrTrack,
                    Vector<Track>(Vector<PrTrack> const&)>
{
     using ScalarTransformer::operator();
     Track operator()(PrTrack const&) const;
};
```

not virtual, will be inlined!

- Future work:
  - recognize vectorized implementation(s), use when appropriate
  - 'zip' : *Vector<I1>,Vector<I2> → vector<O>* from user provided *I1,I2 → O*
  - other python-eque lifting operations: 'product' , 'permutations', 'combinations'

VECTORIZATION — WHERE PERFORMANCE HIDES

- Why does the lifting / elimination of explicit loops matter?
  - Please take a look at this video or these slides
- It matches the several C++ vectorization libraries, eg. Vc, Boost::SIMD, …
- And the C++ Parallelism 2 Technical Specification
- Be Consistent!

```
// parallel and vectorized execution
std::vector<float> v = { ... };
parallel::for_each(
   parallel::datapar_execution,
   std::begin(v), std::end(v),
   [](auto& d) {
      where( d<0, d) = 42.0;
   });
```

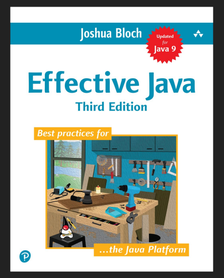
Note the use of a generic lambda

The actual argument type can be a native vector type , or scalar for the stragglers

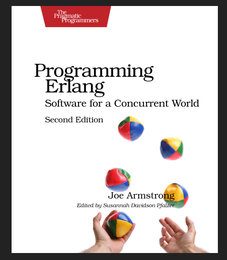
DATA LAYOUT & VECTORIZATION

- Scalable ‘vertical’ vectorization depends on regular, predictable data layout & memory access
- Inheritance makes this impossible
  - Size of objects varies, forcing heap allocation, virtual functions make inlining difficult
- Thin ‘proxies’ (which the compiler should be able to eliminate) can still give *the illusion* of ‘complex objects’
  - see eg. here
  - and provides freedom to optimize the underlying storage to the target hardware (‘AOS’ vs. ‘SOA’ vs ‘AOSOA’ vs …)

EXECUTING COMPONENTS



"Ensure that the average number of runnable threads is not significantly greater than the number of processors."



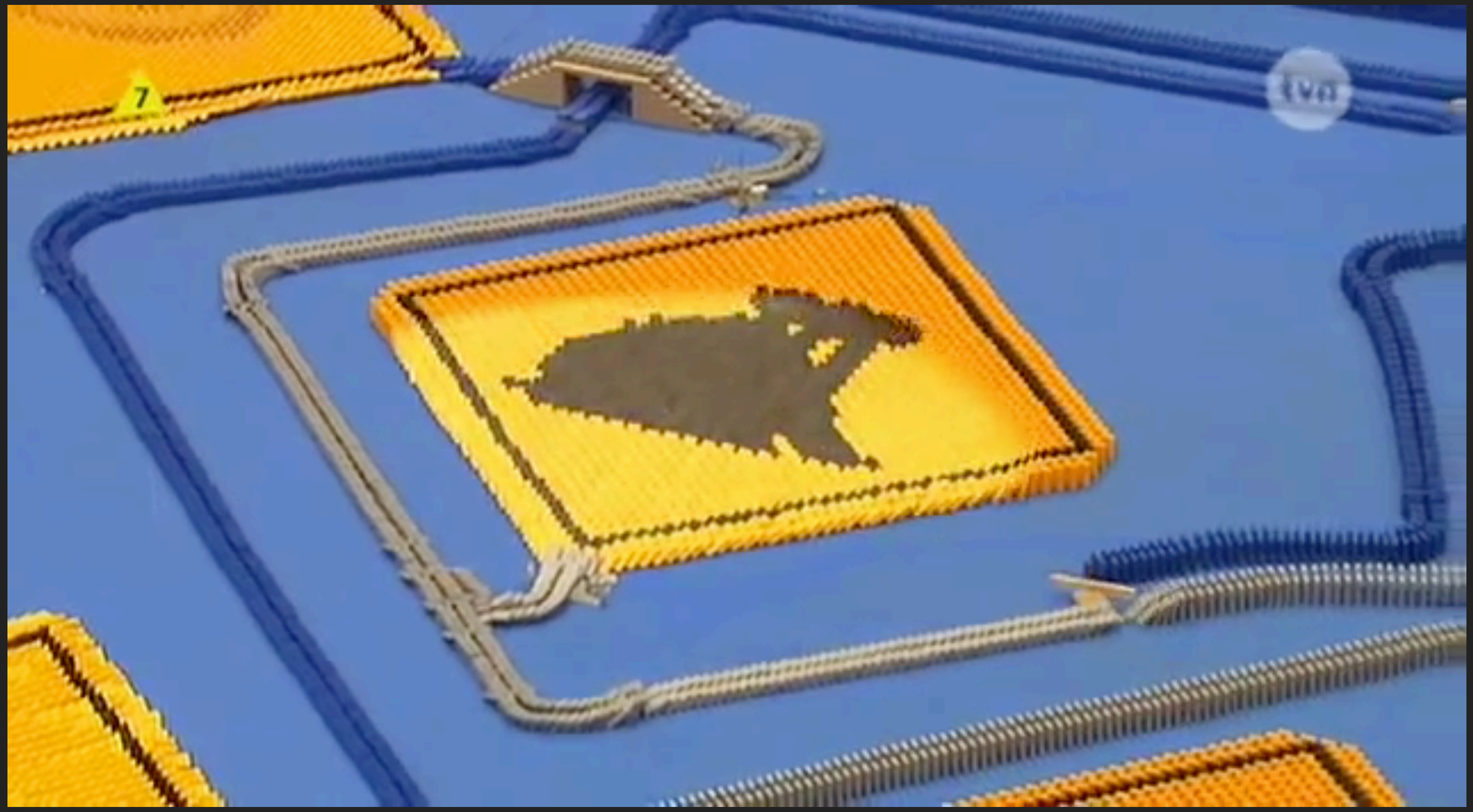
*"**Use Lots of Processes!** This is important—we have to keep the CPUs busy. **All** the CPUs must be busy all the time. The easiest way to achieve this is to have lots of processes. VVhen I say lots of processes, I mean lots in relation to the number of CPUs. If we have lots of processes, then we won't need to worry about keeping the CPUs busy"*

- Erlang process is extremely light-weight: "Erlang is a concurrent programming language – parallelism is provided by Erlang and *not* the host operating system"
- C++: Task systems on top of thread pools (Intel TBB, HPX, MS PPL, Apple GCD, …)
  - Lots of tasks, on top of a pool of threads matching ~ hardware threads

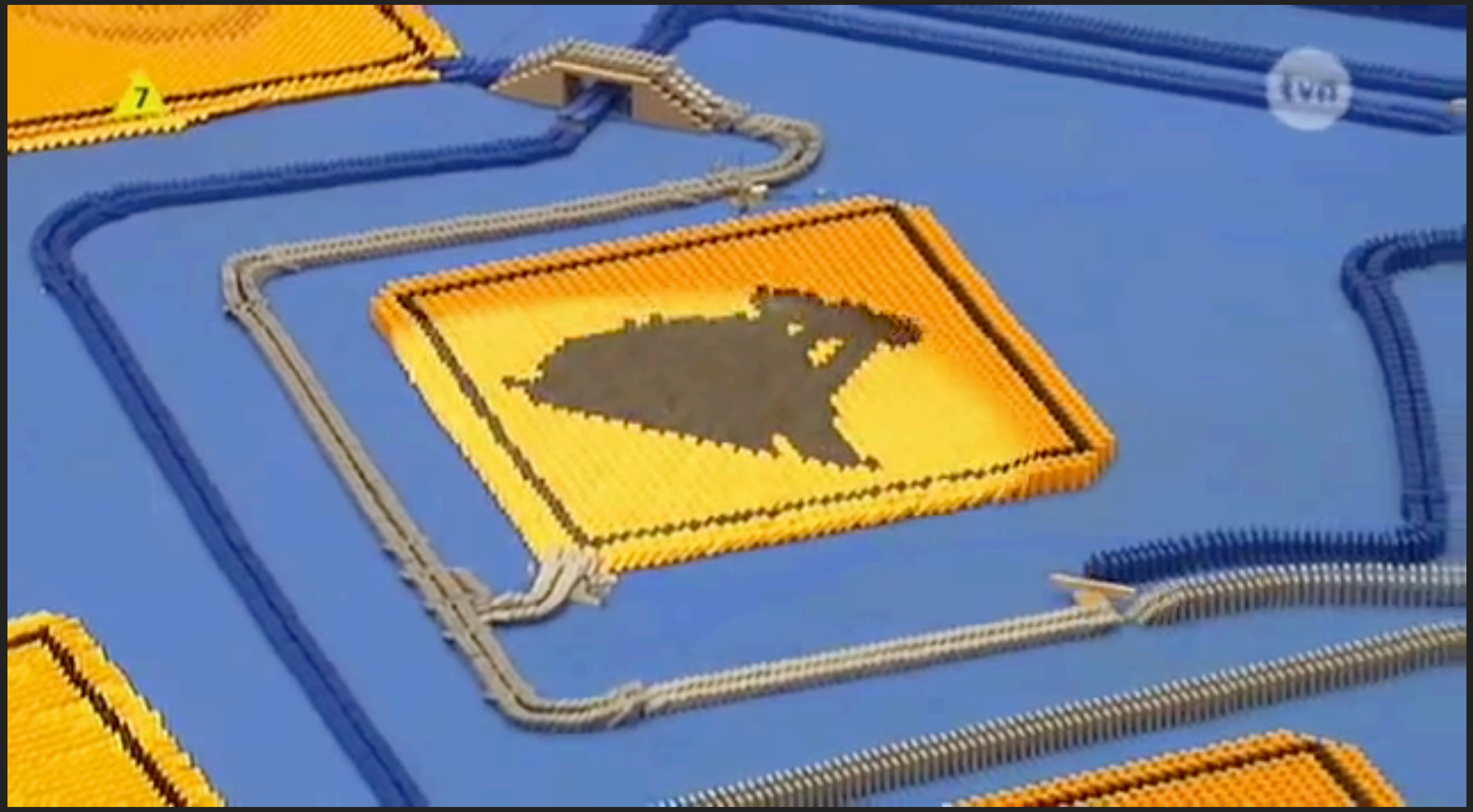
“PEOPLE SHOULD BE ASSIGNED TASKS, AND THEN THEY SHOULD BE EXECUTED”

a senior dutch physicist in a software meeting at CERN in the early 90-ies

DOMINO DAY

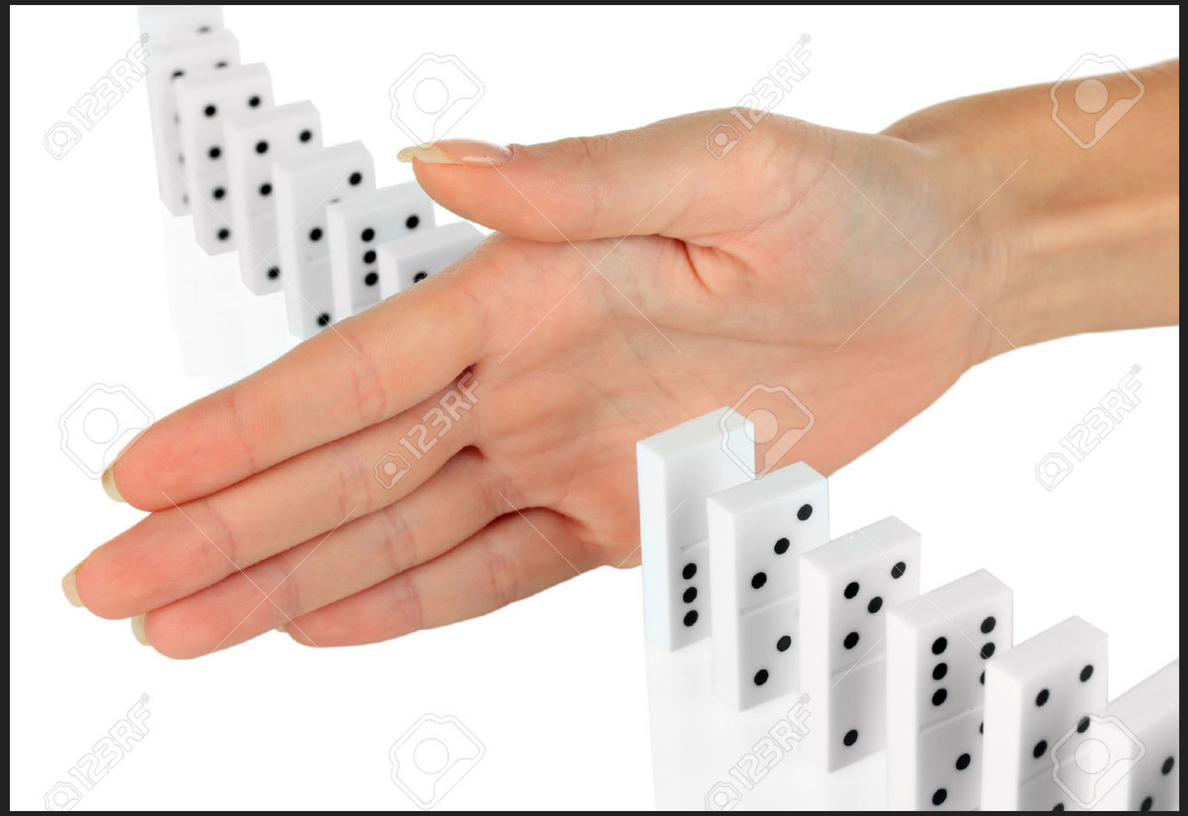


DOMINO DAY



CONTROL FLOW AND DATA FLOW — NOT JUST FOR PARALLELISM!

- Control flow is what 'users' *want* to configure
- Data flow is a 'nuisance parameter' that should be implicit & automatically resolved
  - Using a declarative configuration
- Scheduler infrastructure can help simplify job *configuration*
  - Avoid configuring more than strictly necessary
  - Verification of the validity

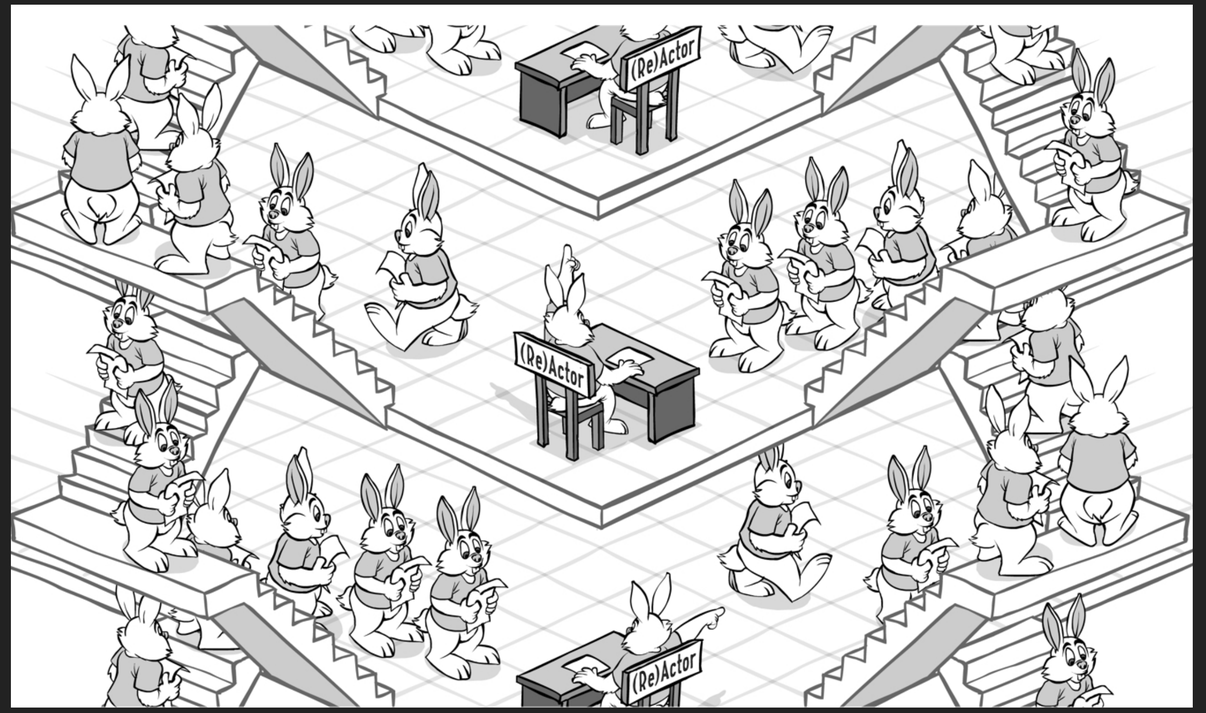


ASIDE — MIGRATIONS

- Migrate code to different paradigms / better abstractions is (very) expensive (short term)
  - it takes effort & time
- But very worthwhile (long term)
  - it simplifies the ‘user code’
    - easier to understand, improve, maintain
  - it simplifies/liberates the framework code!
    - easier to understand, improve, maintain

SUMMARY / CONCLUSIONS

- Learn lessons from the past – and move forwards
- Parallelism is hard, but *necessary* given the hardware landscape
- To do parallelism really right, requires re-imaging – not just a make-over
- Strive for minimal abstractions which allow & enable future evolution
- Data layout and access is key to performance
  - avoid inheritance in data, favor composition, immutable shared data



TALK::~TALK()